J Vis (2024) 27:585–602
https://doi.org/10.1007/s12650-024-00979-x

REGULAR PAPER

Desheng Sun · Xiaoqi Yue · Chao Liu · Hongxing Qin · Haibo Hu

SFLVis: visual analysis of software fault localization

Received: 28 November 2023 / Revised: 6 February 2024 / Accepted: 21 February 2024 / Published online: 2 April 2024
© The Visualization Society of Japan 2024

Abstract Since the birth of software, fault localization has been a time-consuming and laborious task. Programmers need to constantly find faults in software through program logging, assertions, breakpoints, and profiling. In order to improve the debugging efficiency, many fault localization methods based on test cases have been proposed, such as program spectrum-based methods, and slice-based methods. However, these methods are far from the logic of actual debugging and still require programmers to use traditional methods. However, programmers cannot access the execution process of the program, they need to constantly modify breakpoints and repeatedly check variable values, which makes fault localization very time-consuming. After interviewing five experts in the field of visualization and software testing, we designed SFLVis to provide users with a new method to improve the efficiency of fault localization. We designed an algorithm to obtain the process of program execution and combined it with existing fault localization methods. The goal is to show users the execution results of test cases, source code logic, and the level of suspicion of statements, and reproduce the execution process of test cases. We designed rich interactive features to help users explore SFLVis and correlate information from various views to improve the efficiency of fault localization. To verify the effectiveness of SFLVis, we conducted a case study using the program in the Siemens Suite dataset and conducted group experiments and related interviews with 20 volunteers. The results show that SFLVis can effectively improve programmers' efficiency compared with existing fault localization methods.

Keywords Visual analysis · Fault localization · View interaction

D. Sun · X. Yue · C. Liu · H. Hu (✉)
School of Big Data and Software Engineering, Chongqing University, No. 55, University City South Road, Shapingba District, Chongqing 401331, China
E-mail: haibo.hu@cqu.edu.cn

D. Sun
E-mail: ds.sun@cqu.edu.cn

X. Yue
E-mail: xq.yue@cqu.edu.cn

C. Liu
E-mail: liu.chao@cqu.edu.cn

H. Qin
College of Computer Science, Chongqing University, No. 55, University City South Road, Shapingba District, Chongqing 401331, China
E-mail: qinhx@cqu.edu.cn

1 Introduction

Since the first software came into being, software debugging has become an important work. The purpose of software debugging is to discover and repair software faults, including fault localization and fault correction. Fault localization is a time-consuming and tedious task. During the software development process, 50% of the software development and maintenance budget is devoted to fault localization and fault repair (Planning 2002) as no programmers can guarantee that they will write faultless source code (Hao et al. 2009).

The efficiency and accuracy of traditional fault localization methods, such as breakpoints and program logging, often rely on programmers' intuition and experience. To reduce the influence of human factors, many fault localization methods (Abreu et al. 2008); Tip 1994; Binkley and Harman 2004; Xu et al. 2005; Janssen et al. 2009; Abreu et al. 2009; Choi et al. 2010) using causal relationship are proposed (Wong et al. 2016). These methods try to solve the problem of fault localization from different aspects. For example, slice-based methods reduce the search domain for fault localization by deleting irrelevant parts of the code, and program spectrum-based methods help programmers quickly locate faults by calculating the suspicion degree of each statement. With the development of machine learning, some research combines fault localization with machine learning approaches.

Despite the rapid development of fault localization methods, the debugging process of these methods is far from the programmers' traditional debugging process (Hao et al. 2009). These methods only point out potential faulty statements or reduce the scope of localization analysis, which still requires many manual efforts using traditional debugging methods. However, due to the lack of the execution process of the program and the state change of variable values in the execution process, programmers often need to constantly modify breakpoints, repeatedly check variable values, and reduce the scope of the fault according to the prompt, repeating the complex manual process. This condition makes fault localization a time-consuming and tedious task.

To improve the programmers' efficiency in fault localization and reduce manual operation, we discussed with domain experts in software engineering, software testing, and visualization in detail. We created the visual analysis system SFLVis (Visual Analysis of Software Fault Localization).

In SFLVis, we use a heat map to show the execution results of test cases and help users choose the test case for analysis. We use text visualizations to present source code information, capture the actual execution process of the test case, and recreate the program execution process by using text visualizations and heat maps. The node-link diagram is optimized by using the dagre (Sugiyama et al. 1981) layout algorithm to help users understand the code structure. Rich interactions are designed to help users correlate views and explore the system for fault localization. SFLVis allows users to perform fault localization from both an overview view of test cases and the actual execution process of a single test case. It also allows users to make code changes and debug within the system to repair software faults. Together with the domain experts, we demonstrate the system using the Siemens Suite (Do et al. 2005) dataset. Moreover, we recruited 20 volunteers to carry out experiments to verify the effectiveness of the system in fault localization. After the experiment, we obtained their feedback on SFLVis through interviews, which confirmed the effectiveness of SFLVis.

In summary, our contributions are as follows:

- We provide a new method to improve the efficiency of fault localization and implement a visual analysis system SFLVis. The system helps users localize fault through multiple novel visual views and rich interactions between views.
- We design an efficient correlation pipeline to obtain intermediate results of program execution. We design a pipeline to call GDB commands, and directly manipulate GDB through the pipeline to quickly get the corresponding results of the specific process of program execution and related variables.
- We evaluate SFLVis through the case study and user study. In collaboration with domain experts, we conduct the case study using the Siemens suite dataset and evaluate the effectiveness of SFLVis through comparative experiments and interviews with 20 volunteers.

2 Background and related work

Software debugging is an indispensable part (Wong et al. 2005; Pai and Dugan 2007) in the software development cycle, which can effectively reduce software faults and avoid huge losses. Fault localization is

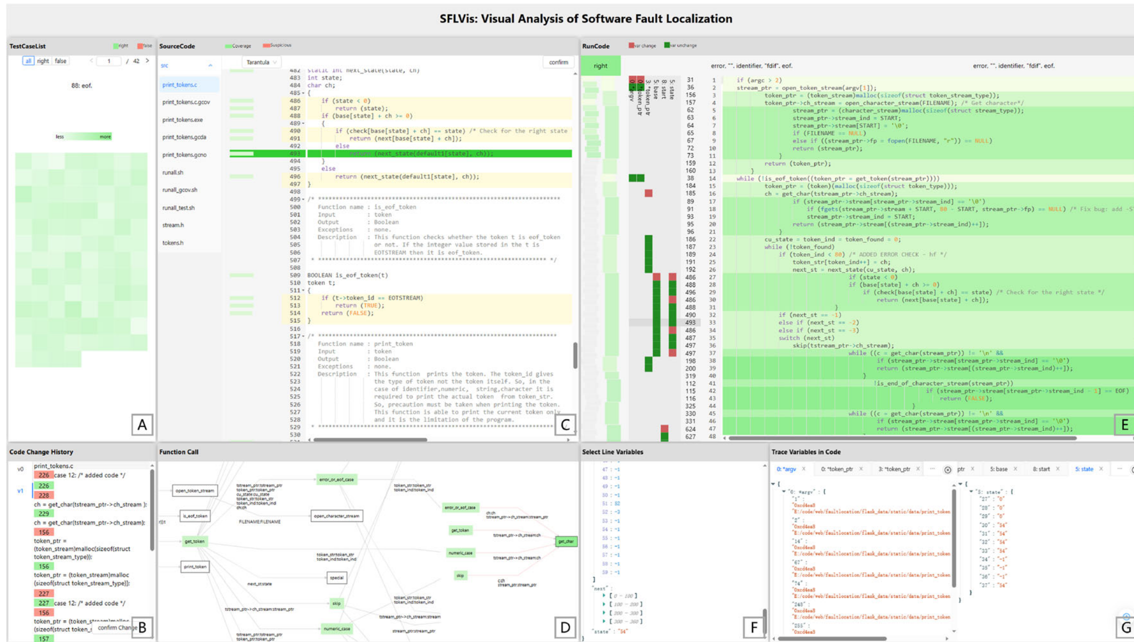

Fig. 1 SFLVis is an interactive visual system that helps users localize software faults with seven interrelated views: **A** Providing an overview of test case execution; **B** Showing the change history of the program; **C** Providing testing results on related source code, such as statement degree of suspicion, test coverage, etc.; **D** Showing program function calls structure; **E** Showing results, statements, functions, and variable changes information about the execution of test case; **F** Displaying the values of the variables related to the execution statement selected by users in the view E; **G** Displaying the results of the variables selected by users in the view G throughout the execution. Users can interactively change the number of display variables in the view

the most time-consuming and tedious task in software debugging. Therefore, improving the efficiency of fault localization can effectively reduce the cost of softwaWe present the source core development. This section introduces the background of fault localization and the related research work.

2.1 Fault localization

Traditional fault localization methods can be subdivided into four schemes: program logging, assertions, breakpoints, and profiling (Wong et al. 2016). Program logging uses statements (such as *print*) to monitor variable values and other program state information, and programmers will use this information to diagnose the cause of the software fault (Edwards 2003). Assertions detect error behavior when a program executes by adding constraints to the program, and when the constraints are true, the correct statement executes Rosenblum 1992; Rosenblum 1995). Breakpoints are used to pause a program and allow the programmers to examine the current state of the program and determine whether the program has faults (Hennessy 1982). Profiling is the program execution speed and memory usage (Hauswirth and Chilimbi 2004), usually used to optimize the program.

However, with the increasing scale of software, the traditional fault localization methods have been unable to find the root cause of the fault effectively (Wong et al. 2016). As a result, many advanced fault localization methods using causality have emerged.

The most extensive of these is the program spectrum-based methods. After all test cases are executed, it analyzes the coverage information for each statement and generates the program spectrum, as shown in Table 1. Finally, the degree of suspicion for each statement is calculated from the program spectrum. Earlier research focused on failed test cases (Collofello and Cousins 1987), but this approach was not considered valid in subsequent studies (Jones and Harrold 2005). Subsequent research compares the difference between successful and failed test cases execution statements (Renieres and Reiss 2003) to see which statements are more suspicious. There are many ways to calculate the degree of suspicion based on the program spectrum (Janssen et al. 2009; Abreu et al. 2009; Choi et al. 2010), and these methods work differently on different programs. These methods provide the programmers with the suspicious value of the statement, and suggest

Table 1 Example program spectrum

	Code with a bug at s_7	a = 0	a = 1	a = 2
s_1	input (a)	•	•	•
s_2	i = 1	•	•	•
s_3	*sum* = 0	•	•	•
s_4	*product* = 1	•	•	•
s_5	if (i < a){	•	•	•
s_6	*sum* = *sum* + *i*			•
s_7	*product* = *product* × *i*			•
	//bug *product* = *product* × *2i*			
s_8	}else{	•	•	
s_9	*sum* = *sum* - *i*	•	•	
s_{10}	*product* = *product* / *i*	•	•	
s_{11}	}	•	•	
s_{12}	print (*sum*)	•	•	•
s_{13}	print (*product*)	•	•	•
	Execution results	S	S	F

the programmers debug based on the order of the suspicious value. However, because this operation is too different from the programmers' actual debugging logic, it is difficult to apply to the actual process.

Besides, another extensive study is slice-based methods (Tip 1994; Binkley and Harman 2004; Xu et al. (2005). The earliest slicing method was static slicing proposed by Weiser (1979). This method makes all the code containing characteristic variables into one slice and deletes other irrelevant parts of the code to simplify the scope of fault localization. However, this approach makes it difficult to see the execution value and often contains irrelevant statements. Therefore, dynamic slicing (Korel and Rilling 1998) was proposed in the subsequent research. This method by analyzing the execution process of the program, makes a slice of the statements that can indeed affect specific variables. These methods can effectively help programmers reduce the scope of fault localization, but they still require programmers to manually debug and check intermediate variables.

With the development of machine learning methods in recent years, there are some fault localization methods combined with machine learning (Cellier et al. 2008; Nessa et al. 2009; Wong and Qi 2009; Naish et al. 2011; Zhang and Zhang 2014). These methods build the neural network (Wong and Qi 2009), input the coverage of a set of virtual test cases into the network that each virtual test case covers only one statement in the program, and treat the output as the likelihood that each statement contains the bug. However, like the program spectrum-based methods, these methods differ greatly from the actual operation process. Moreover, the accuracy of the machine learning methods is difficult to guarantee and even may cause negative effects, so it is difficult to apply in practice.

Although these fault localization methods have become more and more advanced, they still need manual debugging (Wong et al. 2016) by programmers because it is far from the actual fault localization logic. Because of the lack of program execution, the programmers need to repeat the process of modifying the breakpoints and checking variable values.

2.2 Related work

Due to the limitations of fault localization methods, they showed unsatisfactory performance in the actual development process. Therefore, some researchers try to combine fault localization methods with visualization using color coding, visual charts, and other methods to display fault localization results more intuitively.

Due to technical limitations, early visualization systems are simple. Most methods color-coded the source code. For example, James et al. (2002) leveraged visual methods to help programmers with fault localization. This method collects execution results and execution statements for all test cases. Each statement is color-coded and brightly-coded according to the accuracy and coverage of test cases. Later, Orso et al. (2004) implemented GAMMATELLA visualization tools to help programmers collect program execution data, and store and retrieve local data.

With the development of visualization, the visualization methods of fault localization are no longer limited to coding the source code. Hao et al. (2009) proposed VIDA. Based on the analysis of execution information and the feedback collected from the programmers, this method continuously recommends breakpoints for the programmers and records historical breakpoints to help the programmers debug. Jose

et al. proposed the GZoltar tool (Campos et al. 2012), which used a novel constraint-based approach to minimize the original test suite while still guaranteeing the same code coverage. It also provides users with possible locations for suspicious statements. Xie et al. (2018) help users detect exceptions during program execution by improving the behavior of detecting exceptions during execution. Xiao-Yi Zhang et al. proposed the SPICA (Zhang and Jiang 2021), a method that used spectral visualization to review existing SBFL works. Besides, some troubleshooting tools use visualization methods. For example, Ribeiro (2016) (Java coverage fault localization Ranking) colors statements to show the suspicious degree, CodeForest Mutti (2014) uses 3D visual representation in cactus forest to help programmers debug activities and show the most suspicious elements in the program.

Some research helps programmers understand source code visually for fault localization. For example, Abdul et al. constructed FineCodeAnalyzer (Qayum et al. 2022) tool to help programmers analyze source code and locate faults faster through code structure and historical relationship. Tetsuya et al. constructed didiffff (2022) visualization system that helps programmers find potential problems in the program by comparing and visualizing the difference between two execution traces caused by code changes. Nadim et al. (2022) utilized the relational properties of source code in the form of a graph to identify Just-in-Time (JIT) bug prediction in software systems during different revisions of software evolution and maintenance.

In addition, other studies use visualization to understand fault localization methods. Zhang and Zheng (2019) further analyzed the program spectrum-based methods based on the visualized metric analysis framework, and Silva et al. (2018) evaluated the effectiveness and efficiency of the visual tools.

Although the visualization of fault localization has developed rapidly, these methods still require the programmer to constantly modify breakpoints and repeatedly check variable values. Existing visualizations show little of the intermediate process of program execution. Our system displays this information to help programmers quickly understand the running state of the program, to improve the efficiency of fault localization.

3 Requirement analysis

By referring to relevant literature on fault localization methods and fault localization visualization methods, and discussing and cooperating with experts in the corresponding domain, we propose three analysis tasks for the system and formulate the corresponding requirements according to these tasks.

3.1 Task abstraction

In order to construct our analytical task, we investigated papers in several journals. We find the limitations of fault localization methods from TSE (Transactions on Software Engineering), TVCG (Transactions on Visualization and Computer Graphics), etc. Traditional methods still require manual testing by programmers. Due to the lack of a program execution process, programmers need to repeat the same operation, making fault localization time-consuming. Therefore, we determine the ultimate goal of this paper: providing programmers with a new method to overcome limitations, showing the program execution process for the programmers, and improving the efficiency of fault localization.

After determining the goal, we had a number of discussions with five domain experts working in software engineering, software testing, and visualization. Based on the problems encountered in the real scene, we externalized the abstract tasks and formulated three significant tasks as follows:

- T1: Source code structure understanding and the actual execution of test cases. What is the structure of the source code? How does the current test case execute? Which code is involved in execution?
- T2: Understanding code execution information and selecting suspicious code. Which test case should users select for analysis? Which line of code did not pass the test? Which variable causes the fault? What are the values of the variables after each line of code is executed?
- T3: Examining the impact of code modification and determining whether the fault has been repaired. How does the code change affect the current test case? Will all test cases pass completely after the code modification?

3.2 Requirement analysis

Based on the above three tasks, we discussed with domain experts and formulated the four requirements that SFLVis needs to satisfy:

- R1: Providing code viewing. The system should present the original code content to users, and provide basic information such as statement degree of suspicion and test coverage, based on existing fault localization methods. It also should provide users with structure information such as the call relation of the function in the program to help users quickly find the code that may have problems. (T1)
- R2: Providing an overview of test cases. The system should be able to help users understand the execution of all test cases, such as the number of executed code lines, execution results, and output, to select appropriate test cases for analysis. (T2)
- R3: Providing program execution process reproduction. The system should be able to reproduce the actual execution process of the program and correlate with source code information to quickly locate code context. The system also needs to display function call information at the time of the test case execution and analyze the code based on the call information topologically. The system can display the variable information when the line of code is executed, support users to trace the variable, and view the variable change location, to analyze the fault localization of the program. (T1, T2)
- R4: Providing code modification and test cases re-execution. The system should allow users to modify the source code and record the changes to help users repair the fault. When users confirm the modification, the system should re-execute all test cases to determine whether the current modification is valid and whether the program still has faults. (T3)

The target audience for our system is all programmers. So our system should be intuitive, clear, and easy to use, with no additional learning costs for programmers.

4 The SFLVis system

SFLVis provides a new method to help users improve the efficiency of fault localization. This section describes how we collected data for all test cases, and designed novel visual views, and rich view interactions to help users understand and explore SFLVis.

4.1 Data abstraction and operations

We use the Siemens Suite data (Ghandehari et al. 2013) set and decide to use C/C++ as the system language. Siemens Suite is one of the most commonly used datasets for software fault localization, containing seven programs (lines of code between 150 and 800), each with more than 1000 test cases. These programs are made up of multiple versions (a minimum of 7 and a maximum of 41), each containing more than one fault, which is usually caused by modifying one or more lines of code in the program. Existing approaches of fault localization only focus on the results of test case execution, making it difficult for programmers to understand where the code actually fails. However, in the process of actual fault repair, programmers need to know not only the specific process of code operation but also the change of variable values in order to determine the location of the actual fault. We performed the following processing on the data based on real-world fault repair processes.

4.1.1 Test cases data

Based on the dataset of Siemens Suite, we write automated test code to obtain the actual execution results of each test case according to its data characteristics. We use GCC (GNU Compiler Collection) Stallman et al. (1999) technology to generate gcov (Bhushan and Yadav 2017) files for each test case. In this way, the execution coverage and actual execution statements of each test case are obtained, and the relevant data is obtained by writing automated test code.

Based on the gcov file of test cases, we analyze the actual number of statements executed for each test case and the corresponding program spectrum. We obtain the degree of suspicion for each statement according to different program spectrum-based methods, such as the algorithms of Abreu et al. (2006), Jones et al. (2002), and Naish et al. (2011) in Eqs. (1–3), respectively. The symbol meanings are shown in

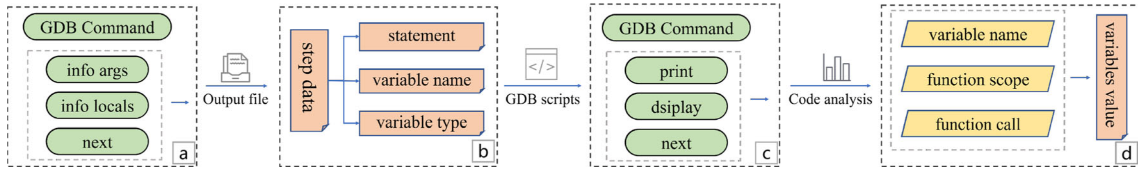

Fig. 2 Use the GDB command to obtain the program execution process, and analyze the data to obtain the detailed information of the program execution

Table 2 The symbol meanings of suspicious degree calculation methods

	Pass	Fail	$\sum$
Execution	N_{ep}	N_{ef}	$N_{ep} + N_{ef}$
Not execution	N_{np}	N_{nf}	$N_{np} + N_{nf}$
$\sum$	N_p	N_f	N

Table 2. These methods behave to varying degrees on different pieces of code. So we calculate a variety of suspicions to provide users, to help them judge.

$$\frac{N_{ef}}{\sqrt{N_f * (N_{cf} + N_{ep})}} \tag{1}$$

$$\frac{N_{ef}/N_f}{(N_{ef}/N_f) + (N_{ep} + N_p)} \tag{2}$$

$$\frac{N_{ef} * N_{np}}{\sqrt{(N_{ef} + N_{ep}) * (N_{np} + N_{nf}) * (N_{ef} + N_{nf}) * (N_{ep} + N_{np})}} \tag{3}$$

We use GDB (GNU symbolic debugger, which is used to debug C/C++ programs) Stallman et al. (1988) to get all the functions defined in the program. We analyze the gcov file to get all the functions that are called during the actual execution. The contents of static files are analyzed to find out the calling relationships between various functions and the passing relationships of variables when these functions are called. This static data can help programmers quickly understand the logical consequences of the program and other relevant information.

4.1.2 Code execution reproduction

In interviews with domain experts, we found that programmers prefer to know the actual execution process of the program and the current values of variables for fault localization. Programmers want to trace suspicious variables and find out where the variables have changed and why. However, few existing fault localization methods involve the actual execution of the program. It is also difficult to obtain actual program execution data when writing automated test scripts with the Siemens Suite dataset.

As the actual execution process of each test case is quite different, it is difficult to write a unified script to obtain its execution process. As a traditional C/C++ compiler debugger, GDB has powerful debugging functions, such as stepping, adding breakpoints, viewing variable values, and so on. Through the analysis of GDB, we implemented a specific pipeline to obtain the actual execution process of test cases as Fig. 2.

We enter a macro command to GDB that allows the program to automatically step through to the end of the program. The command applies to all test cases. When the program executes, each statement has three kinds of variables: global variables, local variables, and function arguments. GDB can obtain all three variables from *info variables*, *info locals*, and *info args*. With these three commands, we complement the macro commands above, which allow the program to get the names of all the current variables in sequence when it steps (Fig. 2a). After executing the macro command, we analyze its output file to get details about the statement that the program has stepped into, the name of the current statement variable, and so on (Fig. 2b). For programmers, it is difficult to debug programs only by executing statements and variable names. GDB can obtain the value of a variable through *print*, *display*, and other commands. For pointer

variables, GDB can only obtain its address, but cannot view the specific value. Therefore, we determine the type of the variable and further trace the pointer variable to obtain its specific value. Then, we write the corresponding script automatically according to the detailed data obtained above and execute the script using GDB. We analyze the output file after GDB executes the script to get the actual execution process of the program and the detailed values of variables after each step (Fig. 2c).

In the actual debugging process, the program fault is often caused by the change of variables. Knowing only the variable values of the current statement, the programmers will not be able to quickly determine where the variables actually changed. We also need to show programmers the details of how each variable changes during execution. We judge the code when the program is executed, analyze the environment variables, functions, and code context of each statement, and judge the variable relationships between different statements. We determine whether the variables between different statements are the same, merge the same variables, and get the statement locations where the variables changed (Fig. 2d).

4.2 Test case view

In discussions with domain experts, we consider that the Test Case View should be simple and intuitive as the user entry point for fault localization. And provide as much information as possible to help users further analyze. So we choose to use a traditional heat map to present information about the test cases (Fig. 1A). We use green codes for successful test cases and red codes for failed test cases and use a gradient from white to the corresponding color to map the number of statements executed for each test case. The closer the color is to white, the fewer statements are executed by the current test case.

The basic heat map can provide little information to users, so we designed different display forms and rich interactions for the Test Case View to provide more information for users (R2). We provide users with a hover interaction to see the output of each test case (Fig. 3a). When users focus on the concrete statement, we highlight all the test cases that execute that statement (Fig. 3b). To help users select an appropriate test case for analysis, we provide selectors and page jumps to change the currently displayed test cases (Fig. 3c). By transitioning the view state, we provide more information without creating visual redundancy for users.

4.3 Source code view

In the existing work, it is difficult to combine the program slice and the statement degree of suspicion with the source code. However, domain experts point out that in the real world, programmers often need to combine multiple pieces of information to determine whether the current statement is actually faulty. But for source code, the text information itself is more important. Therefore, domain experts prefer that we use concise, clear visual expressions to represent the relevant information of the statement.

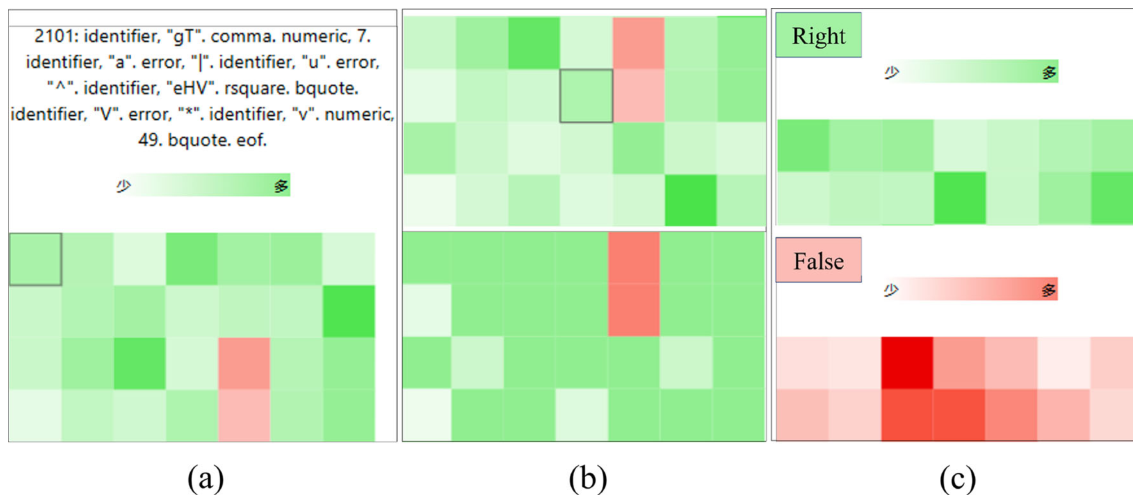

Fig. 3 Test Case View and its presentation in different interactions

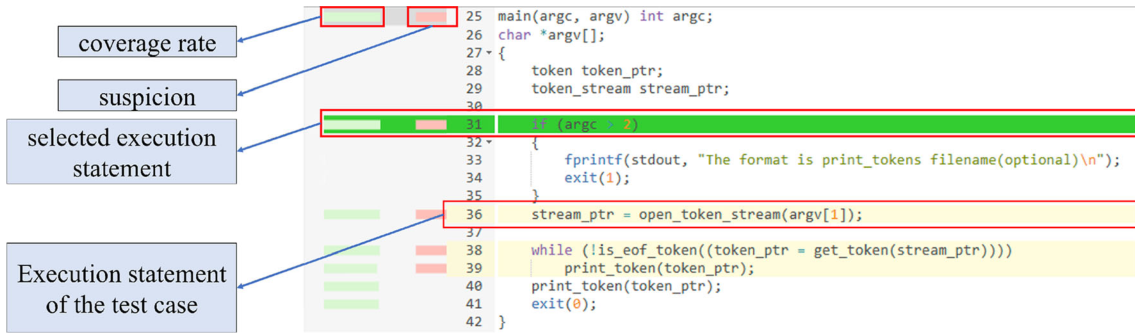

Fig. 4 The Source Code View, and the description of the information shown in each diagram

We present the source code (R1) using text visualizations (Fig. 4), and add length and color-mapped rectangular bars to indicate test case coverage and statement skepticism for the current statement. Because different algorithms get different degrees of suspicion, we provide users with four different calculation methods of suspicion (Jones et al. 2002; Xuan and Monperrus 2014; Abreu et al. 2006; Wong et al. 2012) to assist users to judge. We use a yellow background and a green background to indicate different information about the current statement. To enable users to quickly understand the context of statement execution, when a user selects a statement in the Run Code View, the Source Code View automatically jumps to the page where the statement is located.

Just looking at the source code does not help programmers understand the code logic quickly. We used the dagre layout algorithm to present the static call relationship of the function (R1) to users (Fig. 1D). However, the function call relationship is often very complex and difficult to understand, and the static display presents limited information to users. Thus, we decide to change the state of the view through interaction. When a user selects a statement, the system analyzes the function where the statement resides and draws the call information directly associated with the function (Fig. 5a). The information related to the function is also highlighted on the original view of the Function Call View (Fig. 5b). Users can zoom in or out to change the size of the view to see the details.

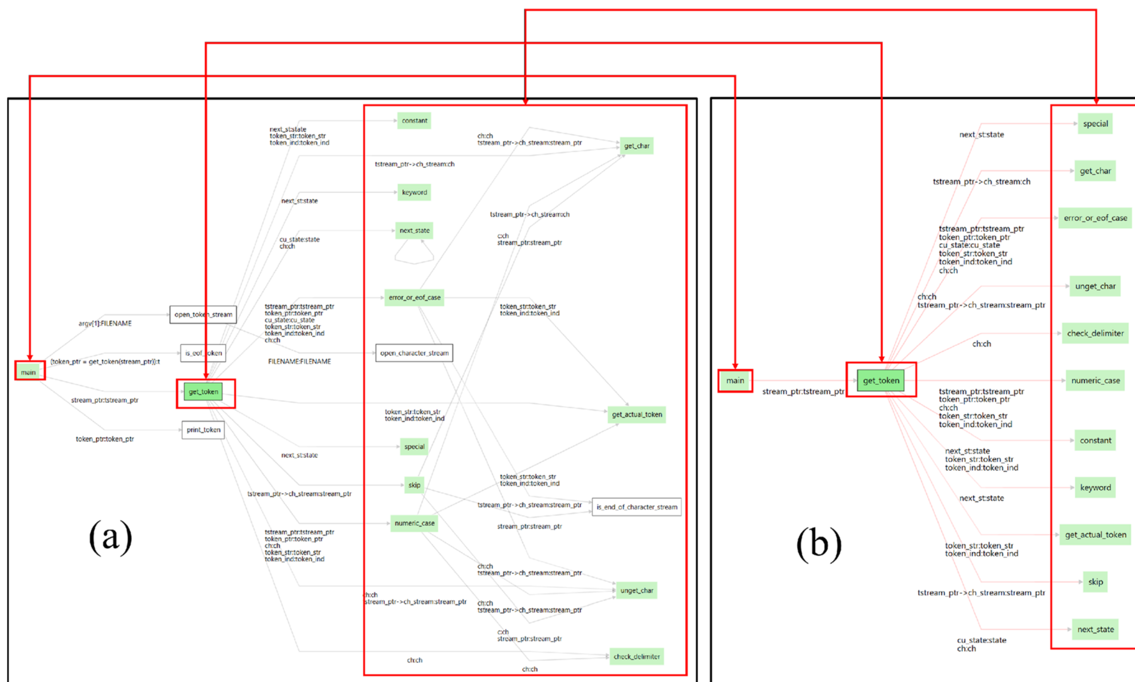

Fig. 5 Function Call View, which changes to view information when users interact

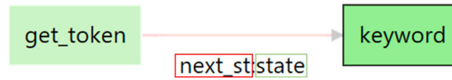

Fig. 6 When a function is called, the variable in the red box is passed to the variable in the green box

Fig. 7 Run Code View: showing the information through four different diagrams

In the Function Call View, we show users the passing of related variables (Fig. 6) when a function is called, so that users can trace the associated related variables in different functions.

4.4 Run code view

Domain experts believe that when debugging in the real world, users need to know not only the execution order of statements but also the values of related variables after each statement execution. At the same time, it allows users to track the variables they are interested in and understand the information such as the change locations of the variables (R3).

Using a single view is difficult to fully display such information at the same time, and will cause cognitive difficulties for users. We use multi-view (Figs. 1E–G) interaction to accomplish these functions.

In the Run Code View, the output of the current test case is shown to users against the correct output through color mapping and text presentation (Fig. 7a). We analyze the function calls of the actual code executed by the test case and show them with thumbnails (Fig. 7b). It also supports users to hover the mouse to view the arguments of the function. The logic of the actual execution is confusing and difficult to understand due to function calls and other situations. So we use function call information to rank each line of statements: the deeper the call level, the lower the statement level. Based on this information, for statements at the same level, we set their backgrounds to the same color coding (Fig. 7d). In Fig. 7c, we use a red rectangle to indicate that the variable value changed during the execution of the statement, and a green rectangle to indicate that the variable value was used during the execution of the statement but did not change. When users select the variable that they are interested in, the system will use different colors in the figure to show the change of the variable.

In Fig. 1F, we show the values of the variables associated with the current statement selected by users. When users are interested in a variable, we allow users to click on that variable for variable tracking. Changes in variables are often caused by other variables, so we also allow multiple variables to be tracked simultaneously. In Fig. 1G, to facilitate users to view variable values and compare different variables, we set a dynamic window to help users track 1-3 variables at the same time.

4.5 Fault localization and debugging

In the above view, users can find the appropriate test case through the Test Case View (Fig. 1A) and choose to execute it. Users can interact with the views of Run Code View (Figs. 1E–G) and Source Code View

(Figs. 1C–D) to find the context information for each statement and the change in the value of each variable. Based on this information, users can determine which statement is faulty.

However, finding the fault is just the beginning of debugging the code, and it is more important to repair the fault. SFLVis provides the ability to change source code and re-execute test cases. However, executing all test cases after each change is time-consuming and exhausting, because users do not know whether they are making the right changes. Based on this situation, we provide the ability to re-execute a single test case (R4). When users modify the source Code and click confirm in the Source Code View (Fig. 1C), SFLVis will re-execute the currently selected test case based on the new code and give a new result. After the modification is complete, users can view the actual code modification in the Code Change History View (Fig. 1B). When users click to confirm the change in the Code Change History View, SFLVis will re-execute all test cases to help users determine whether the fault has been repaired (R4).

5 Evaluation

To evaluate the effectiveness of SFLVis in fault localization, we conducted a case study with domain experts using the Siemens Suite dataset and recruited 20 volunteers for a group experiment.

5.1 Case study

To evaluate the effectiveness and usability of SFLVis, we conducted a case study of a test program in the Siemens Suite. This example shows how SFLVis can help users troubleshoot and its advantages over traditional methods.

Printtokens is a lexical analyzer with seven error versions that can replace and perform pattern matching and substitution. We randomly put three versions of the error into the program for analysis and worked with their domain experts for fault localization. With SFLVis, we ran 4130 of its test cases and found 82 test cases with output that did not match expectations. Combined with the degree of suspicion of each statement in the Source Code View, we decided to analyze the current statement shown in Fig. 8b and found that all of the test cases executing the current statement did not produce the expected result. After analyzing the output results of these test cases and the number of statements executed, we initially selected a test case (the test case is shown in the green box in Figs. 8a–c) with fewer execution times for analysis.

From the Source Code View, we found all the statements that execute the output and the functions that those statements are in. Using the Function Call View, we found the call relationships (Fig. 9a) associated with these functions and the statements associated with calling these functions. In the Run Code View, we traced the variable *token_ptr* (Fig. 9b) by interacting with graphs Figs. 1F–G, found all statements related to *token_ptr* (Fig. 9 c), and the position where the variable has changed. The logic of the change was analyzed to judge the cause of the actual problem.

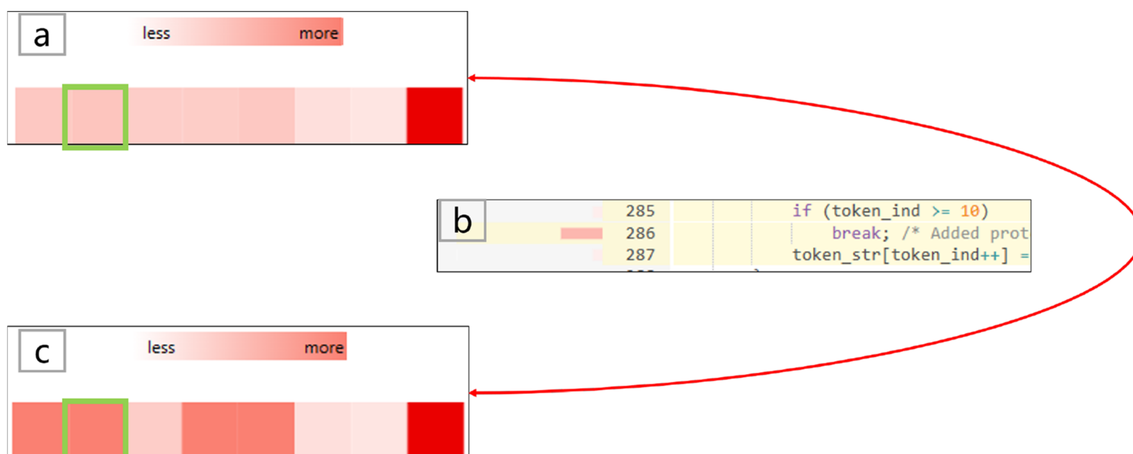

Fig. 8 Test Case View is associated with the degree of suspicion of each statement in Source Code View

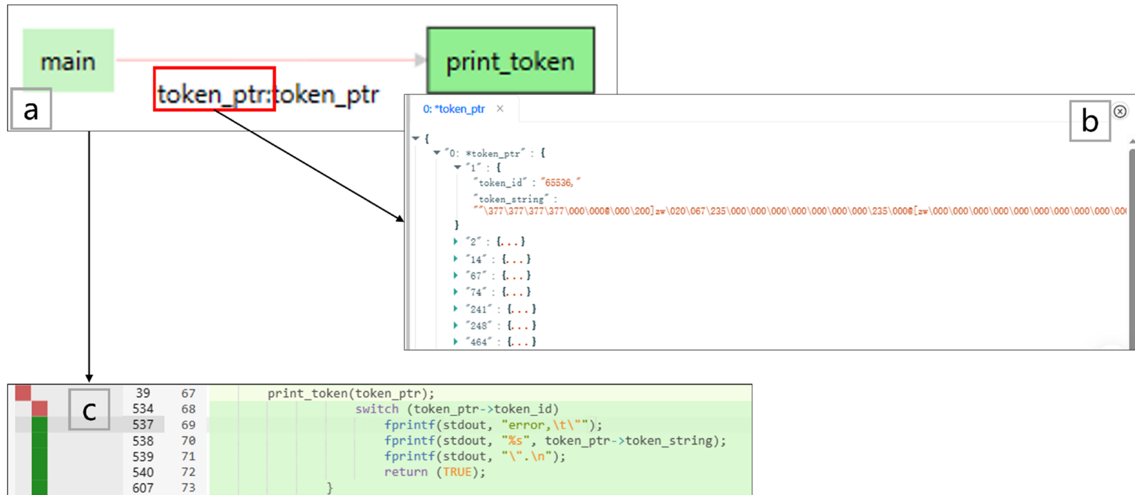

Fig. 9 The execution output statement, call relationships, related variables, and actual execution steps of a function

Fig. 10 Actual execution statements that cause the output to be different from what is expected

By comparing this information with the actual output of the test case, we found the execution statement that caused the output to differ from the expected result (Fig. 10). In the Run Code View, the entire execution process of the current test case was shown, so we can quickly see the logic behind the execution of the program to these statements.

Combined with the degree of suspicion of statements in the Source Code View and variable values after each execution of the statement (Figs. 11a–b), we analyzed that the reason for the fault is that the "break" statement in advance leads to incomplete output of the variable. We modified the relevant code based on the context of the statement. After the modification (Fig. 11c), the system re-executed the test case and found that its output was the same as the expected output. We confirmed the change to the current version and re-executed all the test cases, but found that there were still wrong test cases, indicating that the faults in the program had not been fully repaired, so we repeated the above steps.

Later in the troubleshooting process, we found that some of the highly suspected statements also had successful test cases executed. So we did a comparative analysis to determine the difference between two

```
if (token_ind < 80) /* ADDED ERROR CHECK - hf */
{
    token_str[token_ind++] = ch;
    next_st = next_state(cu_state, ch);
}
```

```
print_tokens.c
v0  285 if (token_ind >= 10)
v1  285 if (token_ind >= 80)
```

Fig. 11 The program actually executes the statement content, associates the code, and modifies the information

```
right
228  ch = get_char(tstream_ptr->ch_stream);
229  if (check_delimiter(ch) == TRUE)
230  {
231      token_ptr->token_id = keyword(next_st);
232      unget_char(ch, tstream_ptr->ch_stream);
233      token_ptr->token_string[0] = '\0';
234      return (token_ptr);
235  }
```

```
false
228  ch = get_char(tstream_ptr->ch_stream);
229  if (check_delimiter(ch) == TRUE)
230  {
231      token_ptr->token_id = keyword(next_st);
232      unget_char(ch, tstream_ptr->ch_stream);
233      token_ptr->token_string[0] = '\0';
234      return (token_ptr);
235  }
236  unget_char(ch, tstream_ptr->ch_stream);
237  break;
```

Fig. 12 Comparison of execution of successful and failed test cases for the same statement. The red and green boxes show the difference in their execution statements

test cases executing the same statement (Fig. 12). This information further improved the efficiency of fault localization, and finally, we effectively solved the fault in the program.

However, in the process of analysis, we also found some problems that affect the efficiency of fault localization. Some statements that generate failures have zero coverage and doubt, such as the "switch case" statement (Fig. 13). These statements had effects when they were actually executed, but the compiler put them together with the statements that were actually executed later, resulting in inconsistent results.

```
222  case 6: /* These are all KEYWORD cases. */
223  case 9:
224  case 11:
225  case 13:
226  case 16:
227  case 32:
228      ch = get_char(tstream_ptr->ch_stream);
229      if (check_delimiter(ch) == TRUE)
```

Fig. 13 Special case: the coverage and suspicion degree of the statement that caused the fault are zero

Although SFLVis has some shortcomings, it can still effectively help programmers with fault localization. To analyze faults in Printtokens using traditional means, users need to keep manually setting breakpoints and running them line by line to see if the current statement is running as expected. In addition, when users find that a fault has occurred, it is difficult to go back to previous statements and can only execute the program again to determine. The judgment of variables also requires users to manually input relevant statements, such as *print*, etc. This process is tedious, and if users make mistakes in the middle, the steps need to be redone. And for each test case, it is difficult for users to know which statements were actually executed, or to compare the differences between different test cases executing the same statement. But SFLVis takes more time in the specific process of retrieving code operations and corresponding results of related variables. When users click the test case to view the execution process, they often wait for a longer time (10–60 S).

Compared to traditional methods and some existing visualization methods, such as VIDA (Hao et al. 2009) (which helps users locate faults by recommending breakpoints), GZoltar (Campos et al. 2012) (which uses a constrained approach to minimize test cases and return to users the possible location of the fault), SPICA (Zhang and Jiang 2021) (which uses a spectral approach to analyze existing SBFL methods), FineCodeAnalyzer (Kanda et al. 2022) (returns possible fault locations through code structure and change history), SFLVis displays the intermediate process of code execution and related information and helps users with fault localization based on existing fault locating methods. It frees the programmer from the manual process of constantly modifying breakpoints and checking variables, presents data interactively, and greatly reduces manual manipulation.

5.2 User study

To confirm that SFLVis can improve the efficiency of fault localization for programmers, we recruited 20 volunteers (12 males and 8 females), all of whom are currently enrolled in computer and software engineering-related programs. They have four to seven years of learning experience (average 5.35 years), and all have programming experience in C/C++. We conducted a controlled experiment on them and interviewed them after the experiment.

5.2.1 Quantitative study

To ensure the accuracy of the experiment, we interviewed the participants before the experiment to ensure that they had not been exposed to the content related to the experiment.

We divided the volunteers into four groups (denoted by G1, G2, G3, and G4), with three men and two women in each group. The experiment involves repairing two different programs, each with only one fault. The experimental programs of groups G1 and G3 were the same, and those of groups G2 and G4 were the same. Groups G1 and G2 used Visual Studio Code for fault localization and were assisted by spectrum-based methods (Jones et al. 2002; Xuan and Monperrus 2014; Abreu et al. 2006; and Wong et al. 2012) and slice-based methods (the execution slice of the program). The above methods are consistent with those provided in SFLVis to reduce the influence of different methods on the experiment. Groups G3 and G4 used SFLVis for fault localization. The experiment ends when volunteers complete the repair of both programs or the experimental time reaches three hours. The standard for the completion of the program repair is that the output results of all test cases are the same as expected.

To minimize the influence of other factors, all volunteers were given relevant training before the experiment. Volunteers in the G1 and G2 groups were shown how to execute test cases against the program. However, volunteers need to write additional code to execute test cases during the experiment. To eliminate the influence of this factor, we provide relevant codes for the automatic execution of test cases, as well as related codes for the automatic calculation of coverage and skepticism. It is calculated automatically when users execute the test case to ensure that users do not spend more time calling these functions. For G3 and G4 volunteers, we showed them how to use the SFLVis and explained the information presented by each view before the experiment. In order to exclude the influence of environmental factors, the four experiments were arranged on four different afternoons, from 14:00 to 17:00, and the experiment locations and computers were the same.

In the laboratory, we recorded the time of each volunteer to repair each program, and the result is shown in the figure. Twenty volunteers from four groups completed the task, and all of them found the fault localization and repaired them within the given time. Due to the different experimental procedures, the

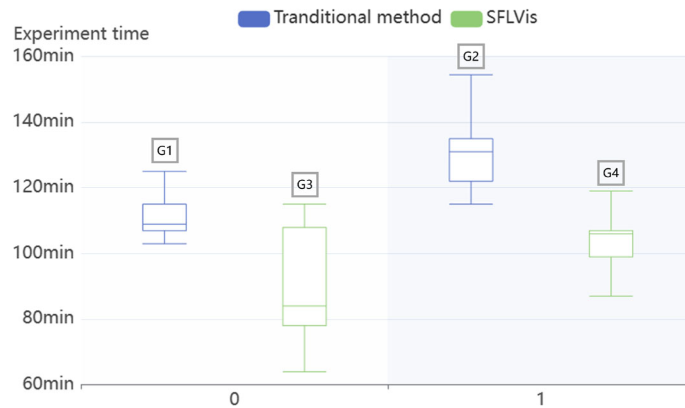

Fig. 14 Grouping experimental results, using the boxplot to represent their highest, lowest, and average values

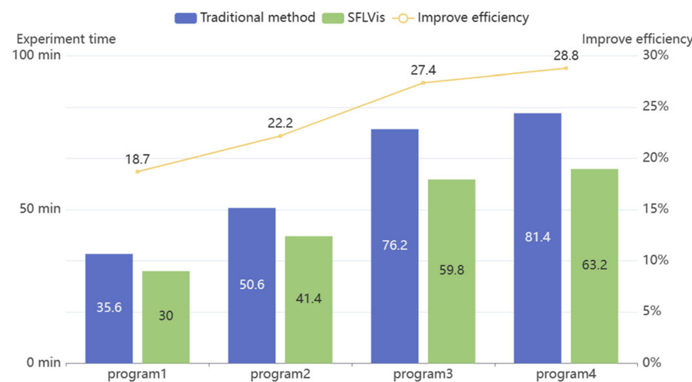

Fig. 15 The test time of each procedure and the percentage of efficiency improvement

experimental time of each group is not the same. However, the experimental time of G3 and G4 using SFLVis was significantly less than that of G1 and G2 (Fig. 14), respectively. The G3 is about 24 percent more efficient than the G1. G4 improved efficiency by about 26 percent compared to G2.

Further analysis showed that the longer the repair program takes, the higher the efficiency of SFLVis (Fig. 15). G1 and G3 had the lowest average time in the first program, and SFLVis improved their efficiency by less than 20 percent. G2 and G4 had the highest average time in the last program, and SFLVis improved their efficiency by more than 28 percent.

5.2.2 *Qualitative study*

During the experiment, we recorded the operation of each volunteer who used SFLVis for fault localization and interviewed them for 10–20 min after the experiment was completed. In the following, V1, V2, ⋯, and V10 will be used for representation (V represents volunteer).

Explore and overview All of the volunteers started with the Source Code View as they conducted the experiment. Once they understood the logic associated with executing the program, they tried to find test cases suitable for analysis based on the view interaction and constantly looked at the statement contents and variable information in the Run Code View. When they found a suspicious statement, they traced the corresponding variable and traced the new related variable according to the Function Call View. Some of the volunteers also looked at the execution of the successful test case and compared it with the execution of the wrong test case. However, the result of this is unpredictable. V1 took significantly less than the average time, but V5 took much more than the average time. In the interview, V5 said that when he looked at the execution results of the successful test case, it was difficult for him to remember the execution process, so he kept switching test cases, which consumed a lot of time. V1 said that comparing the successful test case with

the failed one made him more aware of the logical errors in the wrong test case. In short, we were able to confirm the effectiveness of SFLVis in helping users with fault localization.

Views and interactions In the interview, all the volunteers expressed their recognition of the system. In the process of the experiment, all the views were used, and we found that Run Code View is the most used view, and volunteers spent more than 50 percent of their time operating on Run Code View. Most of the volunteers interacted with Fig. 1F–G constantly. Most volunteers made little or no use of the Code Change History View. V4 said that the Code Change History did not let him know the result of this change, and cannot determine the next change based on this change. V8 said using SFLVis helped him learn more quickly and provided more useful data than traditional fault localization methods. V10 expressed that SFLVis's test case execution process replay was useful, which eliminated the need to constantly execute the program to analyze variables. Showing all the processes in full made it easier for him to understand the program execution logic. V2 indicated that SFLVis was easier to get started with and more acceptable than software testing methods, and rich interactions and simple views made the system easier to explore. But some volunteers have made suggestions of their own. According to V5, the interaction of the system was still imperfect. Users can only view the source code context with Run Code View, but cannot view the relevant information of Run Code View through Source Code View. On the other hand, for complex variables, users still need to analyze by themselves.

SFLVis is well received by the volunteers. The experimental results also show that SFLVis can significantly improve the efficiency of fault localization. In follow-up interviews, volunteers also expressed the advantages of SFLVis compared with traditional schemes. But they also pointed out that some functions of SFLVis were not perfect, which also provided ideas for our subsequent work.

6 Discussion

6.1 Usefulness and limitation

SFLVis has been well received by volunteers, providing users with a new method for fault localization that is significantly more efficient than traditional methods. It helps programmers get out of the routine of setting breakpoints, executing, and viewing variables and keeps users informed of the information they need by reproducing test case execution. By coding the source code, we can also visualize some existing fault localization methods without creating more visual redundancy.

However, SFLVis only supports C/C++ languages at present. Other existing languages, such as Java and JavaScript, cannot implement relevant functions. On the other hand, as the volunteer mentioned earlier, SFLVis needs more functionality. In the process of using SFLVis, there are also some problems, such as difficulty in quickly locating the object code in Run Code View, for complex variables, SFLVis does not have a common query function, and users still need to check manually. After users submit the updated version, the system takes a long time to execute all test cases. This kind of waiting without feedback is long and will lead to incorrect operations by users. In addition, the processing performance of SFLVis for complex programs is uncertain, and we lack research on this aspect. According to our development experience, when large software is developing, it is often divided into different parts for unit testing, and SFLVis can effectively help developers. But when it comes to the overall project, SFLVis will be of limited help.

6.2 Future work

SFLVis is currently a standalone system, but programmers may use integrated development environments such as Visual Studio and Idea with other large compilers. To further increase the versatility of SFLVis, we plan to combine SFLVis-related functionality with these compilers to improve programmers' efficiency. However, SFLVis currently only supports C/C++ languages. We will write relevant algorithms for other languages, such as Java, Python, to obtain the whole process of each execution and corresponding intermediate results. Furthermore, the acquisition of SFLVis intermediate procedures depends on written test cases, but in actual development, it is difficult for users to write relevant test cases in advance. We will try to remove this restriction and allow users to write their own input. The scalability of SFLVis also needs further study. We will optimize the system in future work, use more complex engineering code, and invite more experienced developers to conduct experimental analysis. As mentioned in Sect. 5.2.2, the function of

SFLVis is not perfect enough. We will continue to improve the function of SFLVis according to the suggestions put forward by volunteers.

7 Conclusion

In this paper, we proposed SFLVis which provides a more efficient fault localization solution based on software testing. We designed a correlation algorithm, obtained the actual running process of test cases and corresponding intermediate results, and implemented a visual system to show the process to users. To make it easier for users to explore the system, we used as simple a diagram as possible, designed a rich interaction scheme, and showed how to use SFLVis for fault localization. The effectiveness of SFLVis in fault localization was confirmed by the comparison experiments of 20 volunteers and interviews with them. While they expressed their recognition of SFLVis, they also raised a number of issues that we plan to address in our future work.

Acknowledgements This work was supported by the National Natural Science Foundation of China under Grant 62202074 and U1836114.

References

Abreu R, González A, Zoeteweij P, Gemund AJ (2008) Automatic software fault localization using generic program invariants. In: Proceedings of the 2008 ACM symposium on applied computing, pp 712–717

Abreu R, Mayer W, Stumptner M, Gemund AJ (2009) Refining spectrum-based fault localization rankings. In: Proceedings of the 2009 ACM symposium on applied computing, pp 409–414

Abreu R, Zoeteweij P, Van Gemund AJ (2006) An evaluation of similarity coefficients for software fault localization. In: 2006 12th Pacific rim international symposium on dependable computing (PRDC'06), IEEE, pp 39–46

Bhushan RC, Yadav D (2017) Number of test cases required in achieving statement, branch and path coverage using 'gcov': an analysis. In: 7th international workshop on computer science and engineering (WCSE 2017) Beijing, China, pp 176–180

Binkley DW, Harman M (2004) A survey of empirical results on program slicing. Adv Comput 62(105178):105–178

Campos J, Riboira A, Perez A, Abreu R (2012) Gzoltar: an eclipse plug-in for testing and debugging. In: Proceedings of the 27th IEEE/ACM international conference on automated software engineering, pp 378–381

Cellier P, Ducassé M, Ferré S, Ridoux O (2008) Formal concept analysis enhances fault localization in software. In: Formal concept analysis: 6th international conference, ICFCA 2008, Montreal, Canada, February 25-28, 2008. Proceedings, Springer, 6, pp 273–288

Choi S-S, Cha S-H, Tappert CC et al (2010) A survey of binary similarity and distance measures. J Syst Cybern Inf 8(1):43–48

Collofello JS, Cousins L (1987) Towards automatic software fault location through decision-to-decision path analysis. In: Managing requirements knowledge, international workshop On, IEEE Computer Society, pp 539–539

Do H, Elbaum S, Rothermel G (2005) Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. Empir Softw Eng 10:405–435

Edwards JC (2003) Method, system, and program for logging statements to monitor execution of a program. Google Patents. US Patent 6,539,501

Ghandehari LSG, Bourazjany MN, Lei Y, Kacker RN, Kuhn DR (2013) Applying combinatorial testing to the siemens suite. In: 2013 IEEE Sixth international conference on software testing, verification and validation workshops, IEEE, pp 362–371

Hao D, Zhang L, Zhang L, Sun J, Mei H (2009) Vida: visual interactive debugging. In: 2009 IEEE 31st international conference on software engineering, IEEE, pp 583–586

Hauswirth M, Chilimbi TM (2004) Low-overhead memory leak detection using adaptive statistical profiling. In: Proceedings of the 11th international conference on architectural support for programming languages and operating systems, pp 156–164

Hennessy J (1982) Symbolic debugging of optimized code. ACM Trans Program Languag Syst (TOPLAS) 4(3):323–344

Janssen T, Abreu R, Van Gemund AJ (2009) Zoltar: a spectrum-based fault localization tool. In: Proceedings of the 2009 ESEC/FSE workshop on software integration and evolution@ Runtime, pp 23–30

Jones JA, Harrold MJ (2005) Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the 20th IEEE/ACM international conference on automated software engineering, pp 273–282

Jones JA, Harrold MJ, Stasko J (2002) Visualization of test information to assist fault localization. In: Proceedings of the 24th international conference on software engineering. ICSE, IEEE, pp 467–477

Kanda T, Shimari K, Inoue K (2022)didiffff: a viewer for comparing changes in both code and execution traces. In: Proceedings of the 30th IEEE/ACM international conference on program comprehension, pp 528–532

Korel B, Rilling J (1998) Dynamic program slicing methods. Inf Softw Technol 40(11–12):647–659

Mutti D (2014) Coverage based debugging visualization. PhD thesis, Universidade de São Paulo

Nadim M, Mondal D, Roy CK (2022) Leveraging structural properties of source code graphs for just-in-time bug prediction. Autom Softw Eng 29(1):1–30

Naish L, Lee HJ, Ramamohanarao K (2011) A model for spectra-based software diagnosis. ACM Trans Softw Eng Methodol (TOSEM) 20(3):1–32

Nessa S, Abedin M, Wong WE, Khan L, Qi Y (2009) Fault localization using n-gram analysis. In: Proceedings of the 3rd international conference on wireless algorithms, systems, and applications, pp 548–559
Orso A, Jones JA, Harrold MJ, Stasko J Gammatella (2004) Visualization of program-execution data for deployed software. In: Proceedings. 26th international conference on software engineering, IEEE, pp 699–700
Pai GJ, Dugan JB (2007) Empirical analysis of software fault content and fault proneness using Bayesian methods. IEEE Trans Softw Eng 33(10):675–686
Planning S (2002) The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology, 1
Qayum A, Khan SUR, Akhunzada A et al (2022) Finecodeanalyzer: multi-perspective source code analysis support for software developer through fine-granular level interactive code visualization. IEEE Access 10:20496–20513
Renieres M, Reiss SP (2003) Fault localization with nearest neighbor queries. In: 18th IEEE International conference on automated software engineering, Proceedings, IEEE, pp 30–39
Ribeiro HL (2016) On the use of control-and data-ow in fault localization. PhD thesis, Universidade de São Paulo
Rosenblum DS (1992) Towards a method of programming with assertions. In: Proceedings of the 14th international conference on software engineering, pp 92–104
Rosenblum DS (1995) A practical approach to programming with assertions. IEEE Trans Softw Eng 21(1):19–31
Silva FP, Souza HA, Chaim ML (2018) An empirical assessment of visual debugging tools effectiveness and efficiency. In: 2018 37th international conference of the Chilean computer science society (SCCC), IEEE, pp 1–8
Stallman RM, et al. (1999) Using and porting the GNU compiler collection vol. 86. Free Software Foundation, ???
Stallman R, Pesch R, Shebs S, et al.: Debugging with GDB. Free Software Foundation **675** (1988)
Sugiyama K, Tagawa S, Toda M (1981) Methods for visual understanding of hierarchical system structures. IEEE Trans Syst Man Cybern 11(2):109–125
Tip F (1994) A survey of program slicing techniques. Centrum voor Wiskunde en Informatica Amsterdam, ???
Weiser MD (1979) Program slices: formal, psychological, and practical investigations of an automatic program abstraction method. University of Michigan, USA
Wong WE, Debroy V, Li Y, Gao R (2012) Software fault localization using dstar (d*). In: 2012 IEEE sixth international conference on software security and reliability, IEEE, pp 21–30
Wong WE, Qi Y (2009) BP neural network-based effective fault localization. Int J Softw Eng Knowl Eng 19(04):573–597
Wong WE, Sugeta T, Qi Y, Maldonado JC (2005) Smart debugging software architectural design in SDL. J Syst Softw 76(1):15–28
Wong WE, Gao R, Li Y, Abreu R, Wotawa F (2016) A survey on software fault localization. IEEE Trans Softw Eng 42(8):707–740
Xie C, Xu W, Mueller K (2018) A visual analytics framework for the detection of anomalous call stack trees in high performance computing applications. IEEE Trans Visual Comput Graph 25(1):215–224
Xu B, Qian J, Zhang X, Wu Z, Chen L (2005) A brief survey of program slicing. ACM SIGSOFT Softw Eng Notes 30(2):1–36
Xuan J, Monperrus M (2014) Test case purification for improving fault localization. In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering, pp 52–63
Zhang X-Y, Jiang M (2021) Spica: a methodology for reviewing and analysing fault localisation techniques. In: 2021 IEEE international conference on software maintenance and evolution (ICSME), IEEE, pp 366–377
Zhang X-Y, Zheng Z (2019) A visualization analytical framework for software fault localization metrics. In: 2019 IEEE 24th pacific rim international symposium on dependable computing (PRDC), IEEE, pp 148–14809
Zhang S, Zhang C (2014) Software bug localization with markov logic. In: Companion proceedings of the 36th international conference on software engineering, pp 424–427

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.